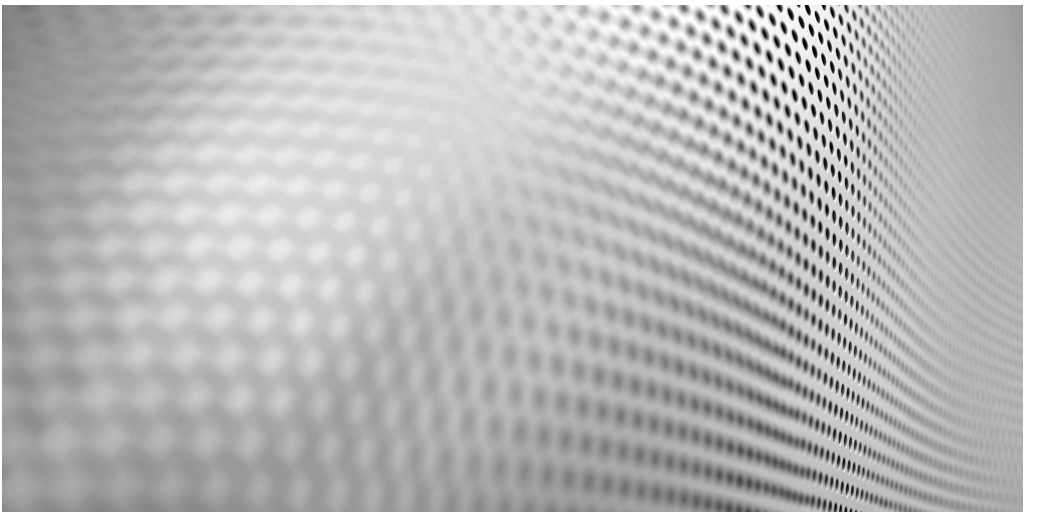

Глава 2

Введение в типы данных и операции

В этой главе...

- Примитивные типы Java
- Использование литералов
- Инициализация переменных
- Правила области видимости переменных внутри метода
- Применение арифметических операций
- Использование операций отношения и логических операций
- Операции присваивания
- Применение сокращенных операций присваивания
- Преобразование типов при присваивании
- Приведение несовместимых типов
- Преобразование типов в выражениях

В основе любого языка программирования лежат типы данных и операции, и Java не является исключением. Типы данных и операции определяют границы применимости языка, а также виды задач, которые можно решать с его помощью. К счастью, в языке Java поддерживается широкий спектр типов данных и операций, что делает его подходящим для написания программ любых категорий.

Типы данных и операторы – обширная тема. Глава начинается с исследования основных типов данных Java и наиболее часто используемых операций. Затем более подробно рассматриваются переменные и выражения.

Важность типов данных

Типы данных особенно важны в Java, поскольку он является строго типизированным языком, т.е. все операции в нем проверяются компилятором на совместимость типов. Код с недопустимыми операциями компилироваться не будет. Таким образом, строгая проверка типов помогает предотвращать ошибки и повышает надежность. Чтобы сделать возможной строгую проверку типов, все переменные, выражения и значения имеют тип. Например, не существует понятия переменной “без типа”. Кроме того, тип значения определяет, какие операции над ним разрешены. Операция, разрешенная для одного типа, может быть запрещена для другого.

Примитивные типы Java

Встроенные типы данных Java подразделяются на две основные категории: объектно-ориентированные и не являющиеся таковыми. Объектно-ориентированные типы определяются классами, а классы будут обсуждаться позже. Однако в основе Java лежат восемь примитивных (также называемых элементарными или простыми) типов данных, которые перечислены в табл. 2.1. Термин *примитивные* применяется для обозначения того, что такие типы – это не объекты в объектно-ориентированном смысле, а обычные двоичные значения. Примитивные типы не реализованы в виде объектов из соображений эффективности. Все остальные типы данных Java созданы из примитивных типов.

Таблица 2.1. Встроенные примитивные типы данных Java

Тип	Описание
`boolean`	Представляет истинные и ложные значения
`byte`	8-битное целое число
`char`	Символ
`double`	Число с плавающей точкой двойной точности
`float`	Число с плавающей точкой одинарной точности
`int`	Целое число
`long`	Длинное целое число
`short`	Короткое целое число

Для каждого примитивного типа Java четко определяет диапазон и поведение, которые должны поддерживаться всеми реализациями виртуальных машин Java. Из-за требования переносимости Java компромисс в таком отношении отсутствует. Например, тип `int` одинаков во всех средах выполнения. В итоге программы могут быть полностью переносимыми. Нет необходимости переписывать код под конкретную платформу. Хотя строгое установление диапазонов для примитивных типов в некоторых средах может привести к небольшому снижению производительности, оно необходимо для обеспечения переносимости.

Целые числа

В Java определены четыре целочисленных типа: `byte`, `short`, `int` и `long`. Все они описаны в табл. 2.2.

Как видно в табл. 2.2, все целочисленные типы представляют положительные и отрицательные значения со знаком. Поддержка только положительных целых чисел без знака в Java отсутствует. Во многих других языках программирования поддерживаются как целые числа со знаком, так и целые числа без знака, но разработчики Java решили, что целые числа без знака не нужны.

Таблица 2.2. Ширина в битах и диапазоны целочисленных типов

Имя	Ширина в битах	Диапазон
byte	8	От –128 до 127
short	16	От –32 768 до 32 767
int	32	От –2 147 483 648 до 2 147 483 647
long	64	От –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807

На заметку!

Формально исполняющая система Java может использовать любой размер для хранения значений примитивного типа. Тем не менее, во всех случаях типы должны действовать так, как указано.

Самым распространенным целочисленным типом можно считать `int`. Переменные типа `int` обычно применяются для управления циклами, для индексации массивов и для выполнения действий из универсальной целочисленной математики.

Если требуется целое число, диапазон которого шире, чем у `int`, тогда нужно использовать `long`. Скажем, вот программа, которая вычисляет количество кубических дюймов, содержащихся в кубической миле:

```
/*
  Вычисляет количество кубических дюймов в кубической миле.
*/
class Inches {
  public static void main(String[] args) {
    long ci;
    long im;
    im = 5280 * 12;
    ci = im * im * im;
    System.out.println("В кубической миле содержится " + ci +
                       " кубических дюймов.");
  }
}
```

Ниже показан вывод, генерируемый программой:

```
В кубической миле содержится 254358061056000 кубических дюймов.
```

Совершенно очевидно, что результат не уместился бы в переменную типа `int`. Наименьшим целочисленным типом является `byte`. Переменные типа `byte` особенно полезны при работе с низкоуровневыми двоичными данными, которые могут быть несовместимыми напрямую с другими встроенными типами Java. Тип `short` позволяет хранить короткие целые числа. Переменные типа `short` подходят, когда нет необходимости в более широком диапазоне, который предлагает тип `int`.

Спросим у эксперта

ВОПРОС. Ранее упоминалось, что есть четыре целочисленных типа: `int`, `short`, `long` и `byte`. Но говорят, что тип `char` в Java тоже можно отнести к категории целочисленных типов. Чем это объясняется?

ОТВЕТ. В формальной спецификации Java определена категория, называемая интегральными типами, куда входят `byte`, `short`, `int`, `long` и `char`. Называние *интегральные* обусловлено тем, что все они предназначены для хранения целых двоичных чисел. Однако первые четыре типа представляют числовые целые величины, тогда как тип `char` — символы. Следовательно, основные способы использования типа `char` и других целочисленных типов фундаментально разные. Из-за таких отличий тип `char` трактуется в книге как отдельный.

Типы с плавающей точкой

Как объяснялось в главе 1, типы с плавающей точкой способны представлять числа с дробной частью. Существуют два типа с плавающей точкой, `float` и `double`, которые представляют числа с одинарной и двойной точностью соответственно. Тип `float` имеет ширину 32 бита, а тип `double` — 64 бита.

Тип `double` применяется чаще типа `float`, и многие математические функции в библиотеке классов Java работают со значениями типа `double`. Например, метод `sqrt()`, определенный в стандартном классе `Math`, возвращает значение типа `double`, которое представляет собой квадратный корень передаваемого методу аргумента типа `double`. Ниже в примере `sqrt()` используется для вычисления длины гипотенузы по длинам двух катетов:

```
/*
  Использование теоремы Пифагора для нахождения
  длины гипотенузы по длинам двух катетов.
*/
class Hypot {
  public static void main(String[] args) {
    double x, y, z;

    x = 3;
    y = 4;

   z = Math.sqrt(x*x + y*y);

 System.out.println("Длина гипотенузы: " +z);
  }
}
```

Обратите внимание на вызов метода `sqrt()`. Он предварен именем класса, членом которого он является.

Вот вывод, генерируемый программой:

```
Длина гипотенузы: 5.0
```

Еще одно замечание по поводу предыдущего примера: как уже упоминалось, метод `sqrt()` является членом стандартного класса `Math`. Обратите внимание на вызов метода `sqrt()`. Ему предшествует имя `Math`, что похоже на то, как `System.out` предшествует `println()`. Хотя не все стандартные методы вызываются с указанием имени класса, для некоторых поступать так удобно.

Символы

В Java символы не являются 8-битными величинами, как во многих других языках программирования; взамен применяется Unicode. Кодировка Unicode определяет полный международный набор символов, с помощью которого можно представить все символы, встречающиеся во всех естественных языках. Тип `char` в Java – это 16-битный тип без знака с диапазоном значений от 0 до 65 535. Стандартный 8-битный набор символов ASCII является подмножеством Unicode и умещается в диапазон от 0 до 127. Таким образом, символы ASCII по-прежнему являются допустимыми символами Java.

Чтобы присвоить значение символьной переменной, символ можно заключить в одинарные кавычки. Например, ниже переменной `ch` присваивается буква `X`:

```
char ch;
ch = 'X';
```

Значение типа `char` выводится с использованием оператора `println()`. Например, следующая строка выводит значение переменной `ch`:

```
System.out.println("Значение ch: " + ch);
```

Поскольку `char` является 16-битным типом без знака, с переменной типа `char` можно выполнять различные арифметические операции. Взгляните на показанную далее программу:

```
// С символьными переменными можно обращаться как с целочисленными.
class CharArithDemo {
  public static void main(String[] args) {
    char ch;

    ch = 'X';
    System.out.println("ch содержит " + ch);

    ch++;     // инкрементирование ch

    System.out.println("ch теперь содержит " + ch);

    ch = 90;  // присваивание ch значения Z
 System.out.println("ch теперь содержит " + ch);
  }
}
```

Переменную типа `char` можно инкрементировать

Переменной типа `char` можно присвоить целочисленное значение

Вот вывод, генерируемый программой:

```
ch содержит X
ch теперь содержит Y
ch теперь содержит Z
```

В программе переменной ch сначала присваивается буква X. Затем значение ch инкрементируется, приводя к тому, что ch станет содержать Y, т.е. следующий символ в последовательности ASCII (и Unicode). Далее ch присваивается значение 90, которое в ASCII (и Unicode) соответствует букве Z. С учетом того, что набор символов ASCII занимает первые 127 позиций в наборе символов Unicode, все "старые трюки", которые вы могли применять в отношении символов в других языках, будут работать и в Java.

Спросим у эксперта

ВОПРОС. Почему в Java используется Unicode?

ОТВЕТ. Язык Java разрабатывался для применения во всем мире. Таким образом, необходимо было использовать набор символов, способный представлять все языки мира. Стандартный набор символов Unicode был разработан специально для этой цели. Разумеется, применение Unicode несколько неэффективно для таких языков, как английский, немецкий, испанский или французский, символы которых могут легко умещаться в пределах 8 бит, но такова цена, которую приходится платить за глобальную переносимость.

Тип boolean

Тип boolean представляет истинные и ложные значения, которые определяются в Java с помощью зарезервированных слов true и false. Таким образом, переменная или выражение типа boolean будет иметь одно из указанных двух значений.

В представленной ниже программе демонстрируется применение типа boolean:

```
// Демонстрация использования значений типа boolean.
class BoolDemo {
  public static void main(String[] args) {
    boolean b;

    b = false;
    System.out.println("b равно " + b);
    b = true;
    System.out.println("b равно " + b);

    // Значение boolean может управлять оператором if.
    if(b) System.out.println("Данная строка кода выполняется.");

    b = false;
    if(b) System.out.println("Данная строка кода не выполняется.");

    // Результатом операции отношения является значение boolean.
    System.out.println("10 > 9 равно " + (10 > 9));
  }
}
```

Программа генерирует следующий вывод:

```
b равно false
b равно true
Данная строка кода выполняется.
10 > 9 равно true
```

В этой программе необходимо отметить три интересных момента. Во-первых, как видите, когда `println()` выводит значение `boolean`, отображается `true` или `false`. Во-вторых, самого по себе значения переменной `boolean` достаточно для управления оператором `if`, т.е. нет необходимости записывать оператор `if` в таком виде:

```
if(b == true) ...
```

В-третьих, результатом операции отношения, подобной <, является значение `boolean`. Именно потому выражение `10>9` дает значение `true`. Кроме того, дополнительный набор скобок вокруг `10>9` нужен из-за того, что приоритет операции + выше приоритета операции >.

Упражнение 2.1 Далеко ли до места вспышки молнии?

`Sound.java`

В данном проекте создается программа, которая вычисляет расстояние (в метрах) между наблюдателем и местом вспышки молнии. Звук распространяется по воздуху со скоростью примерно 335 метров в секунду. Таким образом, зная интервал между моментом, когда появилась вспышка молнии, и моментом, когда был услышан звук, можно рассчитать расстояние до места вспышки. Предположим, что временной интервал составляет 7,2 секунды.

1. Создайте файл по имени `Sound.java`.
2. Имейте в виду, что при расчете расстояния должны использоваться значения с плавающей точкой. Почему? Да потому что временной интервал 7,2 имеет дробную часть. Хотя вполне подошел бы тип `float`, в примере будет применяться тип `double`.
3. Для вычисления расстояния умножьте 7,2 на 335 и присвойте результат переменной.
4. В заключение отобразите результат.

   Ниже показан полный код программы `Sound.java`:

```
/*
  Упражнение 2.1.
  Расчет расстояния до места вспышки молнии,
  звук которого был услышан через 7.2 секунды.
*/
class Sound {
  public static void main(String[] args) {
```

```
        double dist;
        dist = 7.2 * 335;
        System.out.println("Место вспышки молнии находится на расстоянии "
                           + dist + " метров.");
    }
}
```

5. Скомпилируйте и запустите программу. Отобразится следующий результат:

```
Место вспышки молнии находится на расстоянии 2412.0 метров.
```

6. Дополнительная задача: рассчитать расстояние до крупного объекта вроде скалы можно по времени прихода эхо. Например, если вы хлопнете в ладоши и измерите время, через которое стало слышно эхо, то узнаете общее время прохождения звука туда и обратно. Разделив это значение на два, вы получите время, за которое звук проходит в одну сторону. Затем результирующее значение можно использовать для расчета расстояния до объекта. Модифицируйте предыдущую программу так, чтобы она вычисляла расстояние, исходя из предположения о том, что временной интервал представляет собой промежуток времени до прихода эхо.

Литералы

В языке Java под *литералами* понимаются фиксированные значения, представленные в удобочитаемой форме. Скажем, число `100` является литералом. Литералы также обычно называют *константами*. По большей части литералы и их применение настолько интуитивно понятно, что они использовались в той или иной форме во всех предыдущих примерах программ. Теперь пришло время объяснить их формально.

Литералы Java могут относиться к любому примитивному типу данных. Способ представления каждого литерала зависит от его типа. Как объяснялось ранее, символьные константы заключаются в одинарные кавычки, например, `'a'` и `'%'`.

Целочисленные литералы указываются как числа без дробных частей подобно `10` и `-100`. Литералы с плавающей точкой требуют применения десятичной точки, за которой следует дробная часть числа, скажем, `11.123`. Для чисел с плавающей точкой также разрешено использовать экспоненциальную запись.

Целочисленные литералы по умолчанию имеют тип `int`. Чтобы указать литерал типа `long`, понадобится добавить к константе букву `l` или `L`. Например, `12` — литерал типа `int`, а `12L` — литерал типа `long`.

По умолчанию литералы с плавающей точкой имеют тип `double`. Чтобы указать литерал типа `float`, необходимо добавить к константе букву `F` или `f`. Например, `10.19F` — литерал типа `float`.

Хотя целочисленные литералы по умолчанию создаются как значения типа `int`, их все же разрешено присваивать переменным типа `char`, `byte` или `short`, если присваиваемое значение может быть представлено целевым типом. Целочисленный литерал всегда можно присвоить переменной типа `long`.

В целочисленный литерал или литерал с плавающей точкой можно включить один или несколько символов подчеркивания, что позволит облегчить чтение значений, состоящих из многих цифр. Когда литерал компилируется, символы подчеркивания просто отбрасываются. Вот пример:

```
123_45_1234
```

Данный литерал задает значение 123451234. Применение символов подчеркивания особенно полезно при указании номеров деталей, идентификаторов клиентов и кодов состояния, которые обычно состоят из подгрупп цифр.

Шестнадцатеричные, восьмеричные и двоичные литералы

Как известно, в программировании иногда проще использовать систему счисления, основанную на 8 или 16, а не на 10. Система счисления, основанная на 8, называется *восьмеричной*, и в ней применяются цифры от 0 до 7. В восьмеричной системе счисления число 10 соответствует числу 8 в десятичной форме. Система счисления с основанием 16 называется *шестнадцатеричной* и использует цифры от 0 до 9 плюс буквы от `A` до `F`, которые обозначают 10, 11, 12, 13, 14 и 15. Например, шестнадцатеричное число 10 равно десятичному числу 16. Из-за частого применения этих двух систем счисления Java позволяет указывать целочисленные литералы в шестнадцатеричной или восьмеричной форме вместо десятичной. Шестнадцатеричный литерал должен начинаться с `0x` или `0X` (ноль, за которым следует буква `x` или `X`). Восьмеричный литерал начинается с нуля. Ниже приведены примеры:

```
hex = 0xFF;     // соответствует десятичному числу 255
oct = 011;      // соответствует десятичному числу 9
```

Интересно отметить, что в Java также разрешены шестнадцатеричные литералы с плавающей точкой, но они используются редко.

Целочисленный литерал можно указывать с помощью двоичного кода, для чего перед двоичным числом следует поставить `0b` или `0B`. Скажем, `0b1100` задает значение 12 в двоичном формате.

Управляющие последовательности символов

Заключение символьных констант в одинарные кавычки подходит для большинства печатаемых символов, но некоторые символы, такие как возврат каретки, создают особую проблему при работе в текстовом редакторе. Кроме того, ряд других символов вроде одинарных и двойных кавычек имеют специальное предназначение в Java и потому их нельзя применять напрямую. По указанным причинам в Java предусмотрены специальные *управляющие последовательности*,

иногда называемые символьными константами с обратной косой чертой, которые описаны в табл. 2.3. Такие последовательности используются вместо символов, которые они представляют.

Таблица 2.3. Управляющие последовательности символов

Управляющая последовательность	Описание
`\'`	Одинарная кавычка
`\"`	Двойная кавычка
`\\`	Обратная косая черта
`\r`	Возврат каретки
`\n`	Новая строка (также известная как перевод строки)
`\f`	Подача страницы
`\t`	Табуляция
`\b`	Забой
`\ddd`	Восьмеричный символ (`ddd` – восьмеричная константа)
`\uxxxx`	Шестнадцатеричный символ Unicode (`xxxx` – шестнадцатеричная константа)
`\s`	Пробел (последовательность добавлена в JDK 15)
`\конец-строки`	Строка продолжения (применяется только к текстовым блокам; последовательность добавлена в JDK 15)

Следующий оператор присваивает переменной `ch` символ табуляции:

```
ch = '\t';
```

А так переменной `ch` можно присвоить символ одинарной кавычки:

```
ch = '\'';
```

Строковые литералы

В Java поддерживается еще один тип литерала: строка. *Строка* представляет собой набор символов, заключенных в двойные кавычки:

```
"простой тест"
```

Примеры строк встречались во многих операторах `println()` в приведенных ранее программах.

В дополнение к обычным символам строковый литерал может содержать одну или несколько только что описанных управляющих последовательностей. Рассмотрим следующую программу, в которой применяются управляющие последовательности `\n` и `\t`:

```
// Демонстрация использования управляющих последовательностей в строках.
class StrDemo {
  public static void main(String[] args) {
    System.out.println("Первая строка\nВторая строка");
    System.out.println("A\tB\tC");
    System.out.println("D\tE\tF");
  }
}
```

Использовать последовательность \n для вставки символа новой строки

Использовать символы табуляции для выравнивания вывода

Ниже показан вывод, генерируемый программой:

```
Первая строка
Вторая строка
A       B       C
D       E       F
```

Спросим у эксперта

ВОПРОС. Является ли строка, состоящая из одного символа, той же самой сущностью, что и символьный литерал? Например, совпадают ли `"k"` и `'k'`?

ОТВЕТ. Нет. Не путайте строки с символами. Символьный литерал представляет одиночную букву типа `char`. Строка, содержащая только одну букву, по-прежнему считается строкой. Хотя строки состоят из символов, они относятся к разным типам.

Обратите внимание, что для вставки символа новой строки используется управляющая последовательность `\n`. Чтобы получить многострочный вывод, вовсе не обязательно применять несколько операторов `println()`, а нужно просто помещать `\n` в те места более длинной строки, где должны вставляться символы новой строки. И еще один момент: как будет показано в главе 5, недавно в Java появилось функциональное средство, которое называется *текстовым блоком*. Текстовый блок предлагает более высокую степень контроля и гибкости в ситуациях, когда требуется несколько строк текста.

Подробный анализ переменных

Переменные были представлены в главе 1, а здесь они рассматриваются более подробно. Как вы узнали ранее, переменные объявляются с использованием оператора следующего вида:

```
тип имя-переменной;
```

В *тип* указывается тип данных переменной, а в *имя-переменной* — ее имя. Объявлять можно переменную любого допустимого типа, включая только что описанные простые типы, и каждая переменная будет иметь тип. Таким образом, возможности переменной определяются ее типом. Например, переменная типа `boolean` не может применяться для хранения значений с плавающей

точкой. Кроме того, тип переменной не может быть изменен в течение времени ее жизни. Скажем, переменная `int` не может превратиться в переменную `char`.

Все переменные в Java должны быть объявлены до их использования, поскольку компилятору необходимо знать, данные какого типа содержит переменная, прежде чем он сможет правильно скомпилировать любой оператор, в котором применяется переменная. Кроме того, появляется возможность выполнения строгой проверки типов.

Инициализация переменной

Как правило, перед использованием переменной должно быть присвоено значение. Вы уже видели, что один из способов предусматривает применение оператора присваивания. Другой способ – присваивание переменной начального значения при ее объявлении. Для этого после имени переменной ставится знак равенства и нужное значение. Вот общая форма инициализации:

```
тип переменная = значение;
```

В *значение* указывается значение, присваиваемое переменной *переменная* при ее создании. Значение должно быть совместимым с типом, указанным в *тип*. Ниже приведено несколько примеров:

```
int count = 10;         // присвоить count начальное значение 10
char ch = 'X';          // инициализировать ch буквой X
float f = 1.2F;         // инициализировать f значением 1.2
```

При объявлении двух или более переменных одного типа, используя список с разделителями-запятыми, начальное значение можно присваивать одной или нескольким переменным:

```
int a, b = 8, c = 19, d; // переменные b и c имеют инициализаторы
```

В данном случае инициализируются только переменные `b` и `c`.

Динамическая инициализация

Несмотря на то что в предшествующих примерах в качестве инициализаторов применялись только константы, Java позволяет инициализировать переменные динамически с использованием любого выражения, действительного на момент объявления переменной. Скажем, вот короткая программа, которая вычисляет объем цилиндра по радиусу его основания и высоте:

```
// Демонстрация динамической инициализации.
class DynInit {
  public static void main(String[] args) {
    double radius = 4, height = 5;

    // Динамически инициализировать volume.
    double volume = 3.1416 * radius * radius * height;

    System.out.println("Объем составляет " + volume);
  }
}
```

Переменная `volume` динамически инициализируется во время выполнения

Здесь объявляются три локальные переменные: `radius`, `height` и `volume`. Первые две, `radius` и `height`, инициализируются константами, а `volume` динамически инициализируется значением объема цилиндра. Ключевой момент в том, что в выражении инициализации может присутствовать любой элемент, допустимый во время инициализации, в том числе вызовы методов, другие переменные или литералы.

Область видимости и время жизни переменных

До сих пор все применяемые переменные были объявлены в начале метода `main()`. Тем не менее, в Java разрешено объявлять переменные в любом блоке. Как объяснялось в главе 1, блок начинается с открывающей фигурной скобки и заканчивается закрывающей фигурной скобкой. Блок определяет *область видимости*. Таким образом, каждый раз, когда начинается новый блок, создается новая область видимости. Область видимости устанавливает, какие объекты доступны другим частям вашей программы. Она также определяет время жизни этих объектов.

В общем случае каждое объявление в Java имеет область видимости. В результате Java определяет мощную и детализированную концепцию области видимости. Две наиболее распространенных области видимости в Java определяются классом и методом. Обсуждение области видимости класса (и объявленных в нем переменных) откладывается до рассмотрения классов, а сейчас речь пойдет только об областях видимости, определенных методом или внутри него.

Область видимости, определяемая методом, начинается с его открывающей фигурной скобки. Однако если у метода есть параметры, то они тоже входят в область видимости метода. Область видимости метода заканчивается закрывающей фигурной скобкой. Такой блок кода называется *телом метода*.

Как правило, переменные, объявленные внутри области видимости, не будут доступны в коде за рамками этой области. Таким образом, когда вы объявляете переменную в области видимости, то локализуете ее и защищаете от несанкционированного доступа и/или модификации. Действительно, правила области видимости обеспечивают основу для инкапсуляции. Переменная, объявленная внутри блока, называется *локальной переменной*.

Области видимости могут быть вложенными. Например, создавая блок кода, вы создаете новую вложенную область. В таком случае внешняя область видимости охватывает внутреннюю область видимости. В итоге объекты, объявленные во внешней области, будут доступны коду во внутренней области. Тем не менее, обратное утверждение неверно. Объекты, объявленные во внутренней области, не доступны за ее пределами.

Чтобы лучше понять вложенные области видимости, взгляните на следующую программу:

```
// Демонстрация области видимости блока.
class ScopeDemo {
  public static void main(String[] args) {
    int x;          // переменная известна всему коду внутри main()

    x = 10;
    if(x == 10) {   // начало новой области видимости

      int y = 20;   // переменная известна только этому блоку

      // Переменные x и y здесь известны.

      System.out.println("x и y: " + x + " " + y);
      x = y * 2;
    }
    // y = 100; // Ошибка! Переменная y здесь неизвестна

    // Переменная x здесь по-прежнему известна.
    System.out.println("Значение x равно " + x);
  }
}
```

Здесь переменная y находится за пределами своей области видимости

Как указано в комментариях, переменная x объявляется в начале области видимости метода main() и доступна во всем последующем коде main(). Внутри блока if объявляется переменная y. Поскольку блок определяет область видимости, переменная y видна только остальному коду внутри данного блока. Вот почему вне своего блока строка y = 100; закомментирована. Если удалить символы комментария, тогда возникнет ошибка на этапе компиляции, т.к. переменная y не доступна за пределами своего блока. Внутри блока if можно работать с x, потому что код внутри блока (т.е. во вложенной области видимости) имеет доступ к переменным, объявленным в охватывающей области.

Переменные могут быть объявлены в любом месте блока, но они будут действительными только после объявления. Таким образом, если переменная определена в начале метода, то она доступна во всем коде внутри этого метода. И наоборот, переменная, объявленная в конце блока, по существу бесполезна, т.к. никакой код не будет иметь к ней доступа.

Есть еще один важный момент, о котором следует помнить: переменные создаются при входе в свою область видимости и уничтожаются при выходе из этой области видимости. Другими словами, переменная не будет хранить свое значение после того, как покинет свою область видимости, а потому переменные, объявленные внутри метода, не сохраняют значения между его вызовами. Кроме того, переменная, объявленная внутри блока, утрачивает свое значение при выходе из блока. Таким образом, время жизни переменной ограничено ее областью видимости.

Если объявление переменной включает инициализатор, тогда переменная будет повторно инициализироваться каждый раз при входе в блок, в котором она объявлена. Например, рассмотрим показанную ниже программу:

```
// Демонстрация времени жизни переменной.
class VarInitDemo {
```

```
  public static void main(String[] args) {
    int x;

    for(x = 0; x < 3; x++) {
      int y = -1;  // y инициализируется при каждом входе в блок
      System.out.println("Значение y равно: " + y); // всегда выводится -1
      y = 100;
      System.out.println("Теперь значение y равно: " + y);
    }
  }
}
```

Вот вывод, генерируемый программой:

```
Значение y равно: -1
Теперь значение y равно: 100
Значение y равно: -1
Теперь значение y равно: 100
Значение y равно: -1
Теперь значение y равно: 100
```

Несложно заметить, что переменная `y` повторно инициализируется значением `-1` при каждом входе во внутренний цикл `for`. Несмотря на последующее присваивание `y` значения `100`, это значение утрачивается.

С правилами областей видимости Java связана одна удивительная особенность: хотя блоки могут быть вложенными, никакая переменная, объявленная во внутренней области видимости, не может иметь такое же имя, как у переменной, объявленной во внешней области видимости. Например, следующая программа, в которой предпринимается попытка объявить две отдельные переменные с одинаковыми именами, не скомпилируется.

```
/*
  В этой программе предпринимается попытка объявить
  во внутренней области переменную с таким же именем,
  как у переменной, определенной во внешней области.

  *** Программа не скомпилируется. ***
*/
class NestVar {
  public static void main(String[] args) {
    int count;

    for(count = 0; count < 10; count = count+1) {
      System.out.println("Значение count: " + count);

      int count;  // Не разрешено!!!
      for(count = 0; count < 2; count++)
        System.out.println("Программа содержит ошибку!");
    }
  }
}
```

Нельзя объявлять переменную `count`, поскольку она уже определена

Операции

Язык Java обеспечивает развитую среду операций. *Операция* – это символ, который сообщает компилятору о необходимости выполнить определенное математическое или логическое действие. В Java есть четыре основных класса операций: арифметические операции, побитовые операции, операции отношения и логические операции. Кроме того, также определены дополнительные операции, предназначенные для обработки ряда особых ситуаций. В настоящей главе рассматриваются арифметические операции, операции отношения и логические операции, а также операция присваивания. Побитовые и другие специальные операции будут обсуждаться позже.

Арифметические операции

В табл. 2.4 перечислены арифметические операции, которые определены в Java.

Таблица 2.4. Арифметические операции Java

Операция	Описание
+	Сложение (также унарный плюс)
–	Вычитание (также унарный минус)
*	Умножение
/	Деление
%	Остаток от деления
++	Инкремент
--	Декремент

Операции +, –, * и / в Java работают точно так же, как в любом другом языке программирования (впрочем, как и в алгебре). Их можно применять к любому встроенному числовому типу данных, а также использовать для объектов типа `char`.

Хотя действия арифметических операций хорошо известны всем читателям, некоторые особые ситуации требуют пояснений. Прежде всего, не забывайте, что когда операция / применяется к целому числу, остаток от деления будет отброшен; скажем, при целочисленном делении `10/3` дает `3`. Получить остаток от целочисленного деления можно с помощью операции `%`. Например, результатом `10 % 3` будет `1`. Операцию `%` в Java можно применять как к целочисленным типам, так и к типам с плавающей точкой. Таким образом, результат `10.0 % 3.0` также равен `1`. Использование операции получения остатка от деления демонстрируется в следующей программе.

```
// Демонстрация использования операции %.
class ModDemo {
  public static void main(String[] args) {
    int iresult, irem;
    double dresult, drem;
    iresult = 10 / 3;
    irem = 10 % 3;
    dresult = 10.0 / 3.0;
    drem = 10.0 % 3.0;
    System.out.println("Результат и остаток от деления 10 / 3: " +
                       iresult + " " + irem);
    System.out.println("Результат и остаток от деления 10.0 / 3.0: " +
                       dresult + " " + drem);
  }
}
```

Ниже показан вывод, генерируемый программой:

```
Результат и остаток от деления 10 / 3: 3 1
Результат и остаток от деления 10.0 / 3.0: 3.3333333333333335 1.0
```

Как видите, результатом операции `%` является остаток от целочисленного деления и деления с плавающей точкой, равный `1`.

Инкремент и декремент

Описанные в главе 1 операции `++` и `--` являются операциями инкремента и декремента. Как вы увидите, они обладают рядом характеристик, которые делают их довольно интересными. Начнем с рассмотрения того, что делают операции инкремента и декремента.

Операция инкремента добавляет 1 к своему операнду, а операция декремента вычитает 1 из своего операнда. Таким образом, оператор

```
x = x + 1;
```

эквивалентен

```
x++;
```

а оператор

```
x = x - 1;
```

эквивалентен

```
x--;
```

Операции инкремента и декремента могут либо предшествовать операнду (префиксная форма), либо следовать за операндом (постфиксная форма). Скажем, оператор

```
x = x + 1;
```

можно записать так:

```
++x;  // префиксная форма
```

или так

```
x++;  // постфиксная форма
```

В приведенном выше примере нет разницы, какая форма применяется – префиксная или постфиксная. Однако когда операция инкремента или декремента является частью более крупного выражения, то имеется важное отличие. Когда операция инкремента или декремента предшествует своему операнду, то сначала выполняется операция, а затем значение операнда используется в остальной части выражения. Если операция инкремента или декремента находится после своего операнда, то в выражении будет применяться значение операнда до инкрементирования или декрементирования. Рассмотрим следующие операторы:

```
x = 10;
y = ++x;
```

В данном случае переменная `y` получит значение `11`. Тем не менее, если код будет иметь показанный ниже вид:

```
x = 10;
y = x++;
```

то значением переменной `y` окажется `10`. В обоих случаях переменная `x` устанавливается в `11`; разница лишь в том, когда это происходит. Возможность контроля над тем, когда происходит операция инкремента или декремента, обеспечивает значительные преимущества.

Операции отношения и логические операции

Операции отношения и *логические операции* отличаются тем, что первые касаются отношений, которые значения имеют друг с другом, а вторые – способов связи истинных и ложных значений. Поскольку результатами операций отношения являются истинные или ложные значения, они часто используются с логическими операциями, и потому здесь обсуждаются вместе.

Операции отношения перечислены в табл. 2.5.

Таблица 2.5. Операции отношения Java

Операция	Описание
==	Равно
!=	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<=	Меньше или равно

В табл. 2.6 описаны логические операции.

Таблица 2.6. Логические операции Java

Операция	Описание
`&`	И
`|`	ИЛИ
`^`	Исключающее ИЛИ
`||`	Короткозамкнутое ИЛИ
`&&`	Короткозамкнутое И
`!`	НЕ

Результатом операции отношения и логической операции является значение типа `boolean`.

Все объекты в Java можно сравнивать на равенство или неравенство с использованием операций == и !=. Однако операции сравнения <, >, <= и >= можно применять только к типам, которые поддерживают отношение упорядочивания. Таким образом, все операции отношения могут применяться ко всем числовым типам и к типу `char`, но значения типа `boolean` можно сравнивать только на равенство или неравенство, т.к. значения `true` и `false` не считаются упорядоченными. Скажем, `true > false` в Java не имеет смысла.

Для логических операций операнды и результаты имеют тип `boolean`. Логические операции `&`, `|`, `^` и `!` поддерживают основные логические действия И, ИЛИ, исключающее ИЛИ и НЕ в соответствии с таблицей истинности, представленной в табл. 2.7.

Таблица 2.7. Таблица истинности для базовых логических операций Java

p	q	p & q	p | q	p ^ q	!p
false	false	false	false	false	true
true	false	false	true	true	false
false	true	false	true	true	true
true	true	true	true	false	false

В табл. 2.7 видно, что результатом выполнения операции исключающего ИЛИ будет `true`, когда один и только один операнд равен `true`.

В следующей программе демонстрируется работа нескольких операций отношения и логических операций.

```
// Демонстрация работы операций отношения и логических операций.
class RelLogOps {
  public static void main(String[] args) {
    int i, j;
    boolean b1, b2;

    i = 10;
    j = 11;
```

```
    if(i < j) System.out.println("i < j");
    if(i <= j) System.out.println("i <= j");
    if(i != j) System.out.println("i != j");
    if(i == j) System.out.println("Не выполнится.");
    if(i >= j) System.out.println("Не выполнится.");
    if(i > j) System.out.println("Не выполнится.");

    b1 = true;
    b2 = false;
    if(b1 & b2) System.out.println("Не выполнится.");
    if(!(b1 & b2)) System.out.println("!(b1 & b2) дает true");
    if(b1 | b2) System.out.println("b1 | b2 дает true");
    if(b1 ^ b2) System.out.println("b1 ^ b2 дает true");
  }
}
```

Вот вывод, генерируемый программой:

```
i < j
i <= j
i != j
!(b1 & b2) дает true
b1 | b2 дает true
b1 ^ b2 дает true
```

Короткозамкнутые логические операции

В языке Java предусмотрены специальные короткозамкнутые версии логических операций И и ИЛИ, которые можно использовать для создания более эффективного кода. Давайте выясним причину. Если первый операнд операции И равен `false`, тогда результатом будет `false` независимо от того, какое значение имеет второй операнд. Если первый операнд операции ИЛИ равен `true`, то результатом будет `true` независимо от значения второго операнда. Таким образом, в описанных двух случаях нет необходимости вычислять второй операнд, что экономит время и приводит к получению более эффективного кода.

Короткозамкнутая операция И обозначается с помощью `&&`, а короткозамкнутая операция ИЛИ – посредством `||`. Их нормальными аналогами являются операции `&` и `|`. Единственная разница между обычной и короткозамкнутой версией связана с тем, что в нормальных операциях всегда вычисляются все операнды, а в короткозамкнутых версиях второй операнд вычисляется только по мере необходимости.

Далее приведена программа, демонстрирующая работу короткозамкнутой операции И. Она выясняет, будет ли значение `d` множителем `n`, за счет выполнения операции деления по модулю. Если остаток от деления `n / d` равен нулю, то `d` – множитель `n`. Но поскольку операция деления по модулю включает в себя деление, для предотвращения ошибки деления на ноль применяется короткозамкнутая версия операции И.

```
// Демонстрация работы короткозамкнутой операции.
class SCops {
  public static void main(String[] args) {
    int n, d, q;

    n = 10;
    d = 2;
    if(d != 0 && (n % d) == 0)
      System.out.println(d + " - множитель " + n);

    d = 0;  // установить d в ноль

    // Поскольку d равно нулю, второй операнд не вычисляется.
    if(d != 0 && (n % d) == 0) ◄──── Короткозамкнутая операция предотвращает деление на ноль
      System.out.println(d + " - множитель " + n);

    /* Теперь попробовать то же самое, не используя короткозамкнутую
       операцию. В итоге возникнет ошибка деления на ноль.
    */
    if(d != 0 & (n % d) == 0) ◄──── Теперь вычисляются оба выражения, делая возможным деление на ноль

      System.out.println(d + " - множитель " + n);
  }
}
```

Чтобы предотвратить деление на ноль, с помощью оператора `if` сначала проверяется, равно ли `d` нулю. Если это так, тогда короткозамкнутая операция И останавливается в этой точке и не выполняет деление по модулю. Таким образом, в первой проверке `d` равно `2` и операция деления по модулю выполняется. Вторая проверка завершается неудачно, потому что `d` установлено в ноль, а операция деления по модулю пропускается, позволяя избежать ошибки деления на ноль. В конце испытывается обычная операция И, которая приводит к вычислению обоих операндов, в результате чего во время выполнения возникает ошибка деления на ноль.

И последнее замечание: в формальной спецификации Java короткозамкнутые операции называются *условным ИЛИ* и *условным И*, но обычно используется термин *короткозамкнутые операции*.

Операция присваивания

Операция присваивания применялась, начиная с главы 1, так что пора представить ее формально. Операция присваивания обозначается одиночным знаком равенства (=) и работает в Java почти так же, как в любом другом языке программирования. Вот ее общий вид:

```
переменная = выражение;
```

Тип переменной должен быть совместимым с типом выражения.

Операция присваивания обладает одной интересной особенностью, с которой вы, возможно, пока еще не знакомы: она позволяет создавать цепочку присваиваний.

Спросим у эксперта

ВОПРОС. Учитывая, что короткозамкнутые операции в ряде случаев эффективнее своих нормальных аналогов, почему в Java по-прежнему предлагаются обычные операции И и ИЛИ?

ОТВЕТ. В некоторых случаях требуется, чтобы вычислялись оба операнда операции И либо ИЛИ из-за возникающих побочных эффектов. Взгляните на следующий код:

```
// Побочные эффекты могут быть важны.
class SideEffects {
  public static void main(String[] args) {
    int i;

    i = 0;

    /* Здесь i все равно инкрементируется, несмотря на то,
       что условие в операторе if дает false. */
    if(false & (++i < 100))
      System.out.println("Не отображается");
    System.out.println("Оператор if выполняется: " + i);
        // i имеет значение 1

    /* В данном случае i не инкрементируется, поскольку
       короткозамкнутая операция пропускает инкрементирование. */
    if(false && (++i < 100))
      System.out.println("Не отображается");
    System.out.println("Оператор if выполняется: " + i);
        // i по-прежнему имеет значение 1!
  }
}
```

Как указано в комментариях, в первом операторе `if` значение `i` инкрементируется независимо от того, удовлетворено ли условие. Тем не менее, в случае применения короткозамкнутой операции переменная `i` не инкрементируется, если первый операнд имеет значение `false`. Суть в том, что если код ожидает вычисления правого операнда операции И либо ИЛИ, то должны использоваться нормальные, а не короткозамкнутые формы этих операций.

Например, взгляните на следующий фрагмент кода:

```
int x, y, z;
x = y = z = 100;  // устанавливает x, y и z в 100
```

В приведенном фрагменте кода переменные `x`, `y` и `z` устанавливаются в `100` с помощью одного оператора. Прием работает, потому что = представляет собой операцию, которая возвращает значение правого выражения. Таким образом, значение `z = 100` равно `100`, которое и присваивается `y`, а затем `x`. Использование цепочки присваивания является простым способом присвоить группе переменных общее значение.

Сокращенные операции присваивания

В Java предлагаются специальные сокращенные операции присваивания, которые упрощают код отдельных операторов присваивания. Начнем с примера. Показанный ниже оператор:

```
x = x + 10;
```

можно переписать с применением сокращенной операции присваивания:

```
x += 10;
```

Пара операций += указывает компилятору о необходимости присвоить x значение x плюс 10. Рассмотрим еще один пример. Следующий оператор:

```
x = x - 100;
```

эквивалентен такому:

```
x -= 100;
```

Оба оператора присваивают переменной x значение x минус 100.

Сокращение подобного рода будет работать для всех бинарных операций Java (т.е. требующих двух операндов). Вот общая форма сокращения:

```
переменная операция= выражение;
```

Ниже перечислены арифметические и логические сокращенные операции присваивания.

+=	-=	*=	/=
%=	&=	|=	^=

Поскольку эти операции объединяют операцию с присваиванием, их формально называют *составными операциями присваивания*.

С составными операциями присваивания связаны два преимущества. Во-первых, они более компактны, чем их “длинные” аналоги. Во-вторых, в некоторых случаях они более эффективны. По указанным причинам составные операции присваивания часто встречаются в профессионально написанных программах на Java.

Преобразование типов при присваивании

В программировании принято присваивать переменную одного типа переменной другого типа. Скажем, может потребоваться присвоить значение int переменной float:

```
int i;
float f;

i = 10;
f = i;  // присваивает значение int переменной float
```

Когда в операторе присваивания смешиваются совместимые типы, значение правой части автоматически преобразуется в тип левой части. Таким образом, в предыдущем фрагменте значение `i` преобразуется в число типа `float` и затем присваивается `f`. Однако из-за строгой проверки типов в Java не все типы совместимы, а потому не все преобразования типов неявно разрешены. Например, типы `boolean` и `int` несовместимы.

Когда данные одного типа присваиваются переменной другого типа, *автоматическое преобразование типов* происходит, если

- два типа совместимы;
- целевой тип крупнее исходного.

В случае удовлетворения обоих условий происходит *расширяющее преобразование*. Скажем, тип `int` всегда достаточно велик, чтобы хранить все допустимые значения `byte`, к тому же `int` и `byte` являются целочисленными типами, а потому может быть применено автоматическое преобразование из `byte` в `int`.

Для расширяющих преобразований числовые типы, включая целочисленные типы и типы с плавающей точкой, совместимы друг с другом. Например, следующая программа совершенно корректна, т.к. преобразование из `long` в `double` — это расширяющее преобразование, которое выполняется автоматически.

```
// Демонстрация автоматического преобразования из long в double.
class LtoD {
  public static void main(String[] args) {
    long L;
    double D;

    L = 100123285L;
    D = L;  <-- Автоматическое преобразование из long в double

    System.out.println("L и D: " + L + " " + D);
  }
}
```

Несмотря на наличие автоматического преобразования из `long` в `double`, автоматическое преобразование из `double` в `long` не существует, потому что оно не является расширяющим. Таким образом, показанная далее версия предыдущей программы недопустима.

```
// *** Эта программа не скомпилируется. ***
class LtoD {
  public static void main(String[] args) {
    long L;
    double D;

    D = 100123285.0;
    L = D;  // Не разрешено!!!   <-- Автоматическое преобразование из long в double не существует

    System.out.println("L и D: " + L + " " + D);
  }
}
```

Кроме того, отсутствует автоматическое преобразование числовых типов в `char` или `boolean`. Вдобавок типы `char` и `boolean` несовместимы друг с другом. Тем не менее, переменной типа `char` может быть присвоен целочисленный литерал.

Приведение несовместимых типов

Несмотря на удобство автоматического преобразования типов, оно не сможет удовлетворить всем требованиям, поскольку применяется для *сужающих преобразований* между совместимыми типами. Во всех остальных случаях должно использоваться *приведение*, т.е. инструкция компилятору о необходимости преобразования одного типа в другой. Подобным образом запрашивается явное преобразование типа. Приведение имеет следующую общую форму:

```
(целевой-тип) выражение
```

В целевом типе указывается желаемый тип для преобразования заданного выражения. Например, чтобы преобразовать тип выражения `x / y` в `int`, можно записать так:

```
double x, y;
// ...
(int) (x / y)
```

Несмотря на то что `x` и `y` имеют тип `double`, приведение преобразует результат выражения в `int`. Круглые скобки вокруг `x / y` обязательны, иначе приведение к типу `int` применялось бы только к `x`, а не к результату деления. Приведение здесь необходимо, потому что нет автоматического преобразования из `double` в `int`.

Когда приведение включает в себя сужающее преобразование, информация может быть утрачена. Скажем, при преобразовании значения `long` в `short` информация будет теряться, если значение `long` выходит за рамки диапазона `short`, т.к. старшие биты значения `long` удаляются. Когда значение с плавающей запятой преобразуется в целочисленный тип, дробная часть также будет утрачена из-за усечения. Например, если присвоить целочисленной переменной значение `1.23`, то результирующим значением окажется просто `1`, а `0.23` потеряется.

В показанной далее программе демонстрируется ряд преобразований типов, которые требуют приведений.

```
// Демонстрация приведений.
class CastDemo {
  public static void main(String[] args) {
    double x, y;
    byte b;
    int i;
    char ch;

    x = 10.0;
    y = 3.0;
```

```
        i = (int) (x / y);  // приведение double к int
        System.out.println("Целочисленный результат x / y: " + i);

        i = 100;
        b = (byte) i;
        System.out.println("Значение b: " + b);

        i = 257;
        b = (byte) i;
        System.out.println("Значение b: " + b);

        b = 88;  // код ASCII для X
        ch = (char) b;
        System.out.println("Значение ch: " + ch);
    }
}
```

В этом преобразовании произойдет усечение

Здесь информация не теряется. Переменная byte способна хранить значение 100.

На этот раз информация утрачивается. Переменная byte не может хранить значение 257.

Приведение между несовместимыми типами

Вот вывод, генерируемый программой:

```
Целочисленный результат x / y: 3
Значение b: 100
Значение b: 1
Значение ch: X
```

Приведение `(x / y)` к типу `int` в программе вызывает усечение дробной части и потерю информации. Затем, когда `b` присваивается значение `100`, потеря информации не происходит, поскольку переменная типа `byte` способна содержать такое значение. Однако при попытке присвоить `b` значение `257` информация утрачивается, т.к. `257` превышает максимальное значение типа `byte`. Наконец, для присваивания значения `byte` переменной `char` информация не теряется, но требуется приведение.

Старшинство операций

В табл. 2.8 показан порядок старшинства операций Java от самого высокого до самого низкого. Многие из перечисленных операций будут обсуждаться позже в книге. Хотя `[]`, `()` и `.` формально являются разделителями, они также могут действовать подобно операциям и в этом качестве обладать наивысшим приоритетом.

Таблица 2.8. Старшинство операций в Java

Высший приоритет						
++ (постфиксная)	-- (постфиксная)					
++ (префиксная)	-- (префиксная)	~	!	+ (унарная)	– (унарная)	(приведение-типа)
*	/	%				

Окончание табл. 2.8

Высший приоритет					
+	-				
>>	>>>	<<			
>	>=	<	<=	instanceof	
&					
==	!=				
^					
|					
&&					
||					
?:					
->					
=	*операция=*				
Низший приоритет					

Упражнение 2.2

Отображение таблицы истинности для логических операций

`LogicalOpTable.java`

Втекущем проекте будет создана программа, отображающая таблицу истинности для логических операций Java. Колонки в таблице должны быть выровненными. В проекте используется несколько функциональных средств, описанных в главе, в том числе одна из управляющих последовательностей Java и логические операции. Кроме того, иллюстрируются различия в приоритетах между арифметической операцией + и логическими операциями.

1. Создайте файл по имени `LogicalOpTable.java`.
2. Чтобы обеспечить выравнивание колонок, будет применяться управляющая последовательность `\t` для встраивания символа табуляции в каждую строку вывода. Например, следующий оператор `println()` отображает заголовок таблицы:

```
System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");
```

3. В каждой последующей строке таблицы с помощью символов табуляции будет позиционироваться результат выполнения операции под соответствующим заголовком.
4. Наберите приведенный ниже код программы `LogicalOpTable.java`:

```
// Упражнение 2.2. Вывод таблицы истинности для логических операций.
class LogicalOpTable {
```

```
  public static void main(String[] args) {
    boolean p, q;
    System.out.println("P\tQ\tAND\tOR\tXOR\tNOT");
    p = true; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
    p = true; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
    p = false; q = true;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
    p = false; q = false;
    System.out.print(p + "\t" + q +"\t");
    System.out.print((p&q) + "\t" + (p|q) + "\t");
    System.out.println((p^q) + "\t" + (!p));
  }
}
```

Обратите внимание на круглые скобки, окружающие логические операции внутри операторов `println()`. Они необходимы из-за старшинства операций Java. Операция + имеет более высокий приоритет, нежели логические операции.

5. Скомпилируйте и запустите программу. Отобразится следующая таблица.

```
P       Q       AND     OR      XOR     NOT
true    true    true    true    false   false
true    false   false   true    true    false
false   true    false   true    true    true
false   false   false   false   false   true
```

6. Попробуйте самостоятельно модифицировать программу, чтобы в ней использовались и отображались значения `1` и `0`, а не `true` и `false`. Задача может потребовать немного больше усилий, чем кажется на первый взгляд!

Выражения

Операции, переменные и литералы являются составными частями *выражений*. Вероятно, вам уже известна общая форма выражения из прошлого опыта программирования либо из алгебры. Тем не менее, здесь мы обсудим несколько аспектов выражений.

Преобразования типов в выражениях

В выражении разрешено смешивать данные двух или более отличающихся типов, если они совместимы друг с другом. Скажем, в выражении можно смешивать данные `short` и `long`, потому что они оба являются числовыми типами. Когда в выражении смешиваются данные разных типов, все они преобразуются в один и тот же тип, что достигается за счет применения так называемых *правил повышения типов* Java.

Первым делом все значения `char`, `byte` и `short` повышаются до `int`. Затем, если один операнд имеет тип `long`, то все выражение повышается до `long`. Если же один операнд относится к типу `float`, тогда все выражение повышается до `float`. Если какой-либо из операндов имеет тип `double`, то результатом будет значение `double`.

Важно понимать, что повышение типов применяется только к значениям, над которыми выполняются операции при вычислении выражения. Например, даже если значение переменной типа `byte` было повышено до `int` внутри выражения, то за пределами выражения типом переменной по-прежнему остается `byte`. Повышение типа влияет только на вычисление выражения.

Однако повышение типов может приводить к несколько неожиданным результатам. Скажем, когда арифметическая операция включает два значения `byte`, то вот что происходит. Сначала операнды типа `byte` повышаются до `int`, после чего выполняется операция, дающая результат типа `int`. Таким образом, результат операции с двумя значениями `byte` будет иметь тип `int`. Это не то, что можно было со всей очевидностью предположить. Рассмотрим следующую программу:

```
// Неожиданное повышение типов!
class PromDemo {
  public static void main(String[] args) {
    byte b;
    int i;

    b = 10;
    i = b * b;  // Нормально, в приведении нет нужды.

    b = 10;
    b = (byte) (b * b);  // Требуется приведение!

    System.out.println("i и b: " + i + " " + b);
  }
}
```

В приведении нет необходимости, потому что результат уже повышен до `int`

Здесь для присваивания значения `int` переменной типа `byte` требуется приведение типов!

Как ни парадоксально, при присваивании переменной `i` значения `b*b` приведение не требуется, поскольку во время вычисления выражения тип `b` повышается до `int`. Тем не менее, при попытке присвоить переменной `b` значение `b*b` требуется приведение обратно к типу `byte`! Имейте такую особенность в виду, когда получаете неожиданные сообщения об ошибках несовместимости типов для выражений, которые иначе казались бы совершенно правильными.

Такая же ситуация возникает и при выполнении операций над операндами типа `char`. Например, в показанном далее фрагменте кода приведение обратно к `char` требуется из-за повышения `ch1` и `ch2` внутри выражения до типа `int`:

```
char ch1 = 'a', ch2 = 'b';
ch1 = (char) (ch1 + ch2);
```

Без приведения результатом сложения `ch1` и `ch2` было бы значение `int`, которое нельзя присвоить переменной типа `char`.

Приведения полезны не только при преобразовании между типами во время присваивания. Скажем, взгляните на следующую программу, в которой используется приведение к `double`, чтобы сохранить дробную часть результата; иначе деление было бы целочисленным.

```
// Использование приведения.
class UseCast {
  public static void main(String[] args) {
    int i;
    for(i = 0; i < 5; i++) {
      System.out.println(i + " / 3: " + i / 3);
      System.out.println(i + " / 3 с дробной частью: "
                         + (double) i / 3); System.out.println();
    }
  }
}
```

Вот вывод, генерируемый программой:

```
0 / 3: 0
0 / 3 с дробной частью: 0.0

1 / 3: 0
1 / 3 с дробной частью: 0.3333333333333333

2 / 3: 0
2 / 3 с дробной частью: 0.6666666666666666

3 / 3: 1
3 / 3 с дробной частью: 1.0

4 / 3: 1
4 / 3 с дробной частью: 1.3333333333333333
```

Использование пробельных символов и круглых скобок

Выражение в Java может содержать символы табуляции и пробелы для повышения удобочитаемости. Например, следующие два выражения одинаковы, но второе легче для восприятия:

```
x=10/y*(127/x);
x = 10 / y * (127/x);
```

Скобки увеличивают приоритет содержащихся в них операций (как в алгебре). Применение лишних или дополнительных скобок не вызывает ошибки и не замедляет выполнение выражения. Круглые скобки рекомендуется использовать с целью прояснения точного порядка вычисления и для себя, и для других, кому придется разбираться в программе позже. Скажем, какое из показанных ниже выражений легче для восприятия?

```
x = y/3-34*temp+127;
x = (y/3) - (34*temp) + 127;
```

Вопросы и упражнения для самопроверки

1. Почему в Java для каждого примитивного типа строго определен диапазон и поведение?
2. Что такое символьный тип в Java и чем он отличается от символьного типа, используемого в ряде других языков программирования?
3. Переменная типа `boolean` может иметь любое желаемое значение, потому что любое ненулевое значение является истинным. Так ли это?
4. Напишите одиночный оператор `println()`, выдающий такой вывод:

   ```
   Один
   Два
   Три
   ```

   Напишите строку кода с вызовом метода `println()`, где этот результат выводится в виде одной строки.
5. Что не так с этим фрагментом кода?

   ```
   for(i = 0; i < 10; i++) {
     int sum;
     sum = sum + i;
   }
   System.out.println("Сумма: " + sum);
   ```
6. Объясните разницу между префиксной и постфиксной формами операции инкремента.
7. Покажите, как можно использовать короткозамкнутую операцию И для предотвращения ошибки деления на ноль.
8. До какого типа повышаются типы `byte` и `short` в выражении?
9. Когда в общем случае необходимо приведение?
10. Напишите программу, находящую все простые числа от 2 до 100.
11. Влияют ли избыточные круглые скобки на скорость выполнения программы?
12. Определяет ли блок область видимости?